Coherence Protocol for Transparent Management of Scratchpad Memories in Shared Memory Manycore Architectures

Lluc Alvarez[†§] Lluís Vilanova[†§] Miquel Moreto[†] Marc Casas[†] Marc Gonzàlez[§]
Xavier Martorell[†§] Nacho Navarro[†§] Eduard Ayguadé[†§] Mateo Valero[†§]

[†]Barcelona Supercomputing Center
{first.last}@bsc.es

[§]Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya

Abstract

The increasing number of cores in manycore architectures causes important power and scalability problems in the memory subsystem. One solution is to introduce scratchpad memories alongside the cache hierarchy, forming a hybrid memory system. Scratchpad memories are more power-efficient than caches and they do not generate coherence traffic, but they suffer from poor programmability. A good way to hide the programmability difficulties to the programmer is to give the compiler the responsibility of generating code to manage the scratchpad memories. Unfortunately, compilers do not succeed in generating this code in the presence of random memory accesses with unknown aliasing hazards.

This paper proposes a coherence protocol for the hybrid memory system that allows the compiler to always generate code to manage the scratchpad memories. In coordination with the compiler, memory accesses that may access stale copies of data are identified and diverted to the valid copy of the data. The proposal allows the architecture to be exposed to the programmer as a shared memory manycore, maintaining the programming simplicity of shared memory models and preserving backwards compatibility. In a 64-core manycore, the coherence protocol adds overheads of 4% in performance, 8% in network traffic and 9% in energy consumption to enable the usage of the hybrid memory system that, compared to a cache-based system, achieves a speedup of 1.14x and reduces on-chip network traffic and energy consumption by 29% and 17%, respectively.

1. Introduction

Future generations of multicore and manycore architectures are expected to include a significant number of cores. As an immediate consequence, the memory system that connects these computing elements needs to be revisited in order to satisfy the inherent requirements of future chips and, at the same time, avoid the inefficiencies of current schemes when cores are replicated beyond certain levels [31, 35, 39].

Cache coherent shared memory has traditionally been the favorite memory organization for multicore chips. The major reason for the success of this approach is its high programmability, that is achieved by moving data and keeping it coherent between all the caches of the system without any intervention from the programmer. Unfortunately, the cost of performing these actions becomes an obstacle to scaling up the number of cores, being the primary concerns the power consumption originated in the caches and the increasing amount of coherence traffic in the Network on-Chip (NoC).

Scratchpad memories [9] (SPMs) are a well-known alternative to cache hierarchies. The simplicity of SPMs allows them to serve memory accesses as fast as caches but in a much more power-efficient way and without originating coherence traffic. However, SPMs suffer from poor programmability as the software is responsible of explicitly transferring data and keeping different copies of the data in a coherent state.

A recent trend in the High Performance Computing (HPC) domain is to combine caches and SPMs in the memory hierarchy, forming a *hybrid memory system*. Architectures such as the Cell B.E. [26] or GPUs [21] have successfully adopted this scheme in different forms, but always at the cost of breaking backwards compatibility and imposing complex programming and memory consistency models that move away from the shared memory paradigm. Unfortunately, this approach cannot be easily adopted by shared memory manycores, where backwards compatibility has a big importance and the wide majority of programming models heavily rely on strict memory models and cache coherence protocols [3].

Still, the characteristics of HPC applications [36, 44] make appealing the adoption of the hybrid memory system in shared memory manycores. HPC applications are dominated by strided accesses, which are a natural fit for SPMs because they can be efficiently managed by software thanks to their predictability. Random accesses, on the other hand, are difficult to manage by software because of their unpredictability, so they greatly benefit from the ability of caches to automati-

ISCA'15, June 13-17, 2015, Portland, OR, USA
 ISBN 978-1-4503-3402-0/15/06...$15.00
DOI: http://dx.doi.org/10.1145/2749469.2750411

cally request data and keep it coherent. A promising solution to exploit these characteristics in shared memory manycores is to introduce the hybrid memory system and to give the compiler the responsibility of generating code to manage the SPMs, so that the added programming complexity is not exposed to the programmer. Even though compilers succeed in generating code for the SPMs when the computation is based on predictable memory access patterns [22], in the presence of unpredictable memory accesses they encounter important limitations. Due to the incoherence between the SPMs and the cache hierarchy, the compiler cannot generate code for the SPMs if it cannot ensure that there is no aliasing between two memory references that may target copies of the same data in the SPMs and in the cache hierarchy. This memory aliasing problem greatly restricts the ability of the compiler to generate code for the hybrid memory system in non-trivial cases.

This paper proposes a coherence protocol for hybrid memory systems that allows the compiler to always generate code for the SPMs, even in the presence of memory aliasing hazards. In a hardware/software co-designed mechanism, the compiler identifies memory accesses that may access incoherent copies of data, and the hardware diverts these accesses to the memory that keeps the valid copy. The proposal allows the compiler to always generate correct code to manage the SPMs, so the hybrid memory system can be exposed to the programmer as a shared memory multiprocessor, maintaining the programming simplicity of shared memory models. Results show that the hybrid memory system, compared to cache hierarchies, provides an average speedup of 1.14x and reduces the network traffic and the consumed energy by respective averages of 29% and 17% in a 64-core manycore.

The main contributions of this paper are:

- A hardware coherence protocol for the hybrid memory system that, complemented with compiler support [5, 6], achieves that all memory accesses are served by the cache or the SPM that keeps the valid copy of the data. This solution solves the memory aliasing problem to the compiler.
- Modifications to the architecture of the hybrid memory system so that, together with the coherence protocol, it implements the memory consistency model assumed in most shared memory programming models.
- A detailed evaluation of a manycore architecture with the hybrid memory system, especially focusing on performance, on-chip network traffic and energy consumption.

This paper is organized as follows: Section 2 explains the architecture of the hybrid memory system together with its compiler and runtime support, the coherence problem it exposes and the compiler support for the coherence protocol. Section 3 presents the coherence protocol and the resulting memory consistency model. Section 4 summarizes the proposal and discusses the OS support to maintain backwards compatibility. Section 5 evaluates the resulting system, which is then compared with the related work in Section 6. Finally, Section 7 remarks the main conclusions of this work.

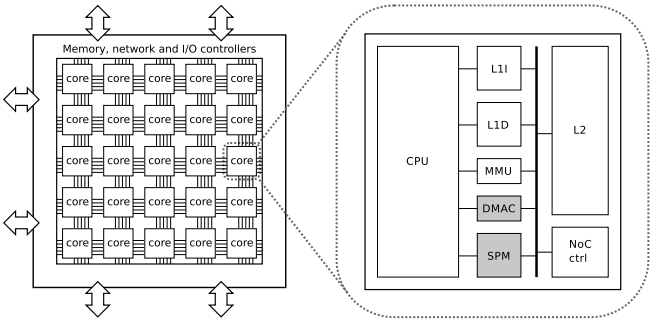

Figure 1: Architecture of the hybrid memory system. Every core is extended with a SPM and a DMA controller (DMAC).

2. Background and Motivation

This section explains the hybrid memory system, its main architectural details and its compiler and runtime support. Then it describes the coherence problem of the architecture and the compiler support for the proposed coherence protocol.

2.1. Hybrid Memory System Architecture

The architecture of the hybrid memory system consists of extending every core of a multiprocessor with a SPM and a DMA controller (DMAC), as shown in Figure 1.

The SPMs are added alongside the L1 cache of every core, and they are accessible by all the cores. The system reserves a range of the virtual and physical address spaces for each SPM of the chip, and direct-maps the virtual ranges to the physical ones, as shown in Figure 2. Every core keeps the address space mapping in eight registers, four to store the starting and the final virtual addresses of the local SPM and of the global range of the SPMs, and four to keep the physical address space of all the SPMs and of the local SPM. These registers are used to identify virtual addresses that access the SPMs and to do the virtual-to-physical address translation, allowing all the cores to access any SPM by issuing loads and stores to their virtual address ranges. At every memory instruction, before any Memory Management Unit (MMU) action takes place, a range check is performed on the virtual address. If the virtual address is in the range of some SPM, the MMU is bypassed and a physical address that points to the SPM is generated. Apart from the simplicity of the implementation, an important advantage of this way of integrating the SPMs [5, 6, 11, 15] is that no pagination is used, so memory accesses to them do not need to lookup the Translation Lookaside Buffer (TLB), minimizing the energy consumption and ensuring deterministic latency. In addition, the typical size of SPMs is orders of magnitude smaller than the size of the RAM and the virtual address space of a 64-bit machine, so the virtual and physical address ranges reserved for the SPMs occupy a very minor portion of the whole address spaces.

The DMACs transfer data between the SPMs and the global memory (GM, which includes caches and main memory). They support three operations: (1) *dma-get* transfers data

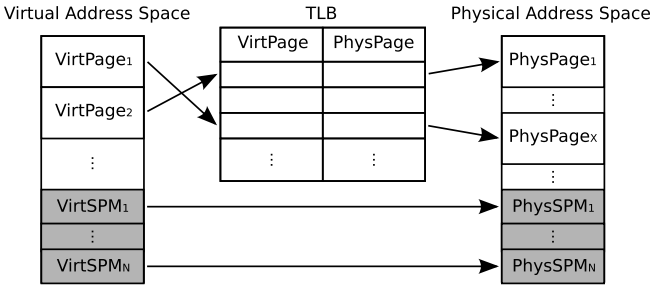

Figure 2: Address space mapping for the SPMs.

from the GM to a SPM, (2) *dma-put* transfers data from a SPM to the GM and (3) *dma-synch* waits for the completion of certain DMA transfers. Every DMAC exposes a set of memory-mapped I/O registers to the software so it can explicitly trigger the DMA operations. DMA transfers are integrated in the cache coherence protocol of the GM [10, 29]. The bus requests generated by a *dma-get* look for the data in the caches and read the value from there if it exists, otherwise they read it from the main memory. The bus requests of a *dma-put* copy the data from the SPM to the main memory and invalidate the cache line in the whole cache hierarchy.

2.2. Compiler and Runtime Support

The main drawback of the hybrid memory system is its programmability, since it is the software that must explicitly manage the SPMs. A good way to hide this complexity from the programmer is to give the compiler the responsibility of generating code that manages the SPMs using a runtime library.

The first step is to identify data suitable to be mapped to the SPMs. In shared memory programming models the compiler uses analyses and code annotations provided by the programmer to decide how the data and the computation is distributed among the threads. Based on this distribution, it identifies array sections [37] that are sequentially traversed and private to each thread. These array sections are the preferred candidates to be mapped to the SPMs because the strided accesses used to traverse them are highly predictable and their privateness avoids costly data synchronization mechanisms on the SPMs.

After identifying the array sections to be mapped to the SPMs the compiler does the code transformations, inserting calls to a runtime library that will manage the data transfers at execution time. For a computational loop the code is transformed into a two-nested loop that uses tiling to do the computation [19, 20, 22, 41], as shown in Figure 3. Each iteration of the outermost loop executes three phases: (1) a control phase that maps chunks of the array sections to the SPMs, (2) a synchronization phase that waits for the completion of the DMA transfers, and (3) a work phase that performs the computation for the currently mapped chunks of data. These phases repeat until the whole iteration space is computed.

Before entering the loop, the runtime divides the size of the SPM in equally-sized buffers in order to minimize complexity and overheads. One buffer is allocated for each

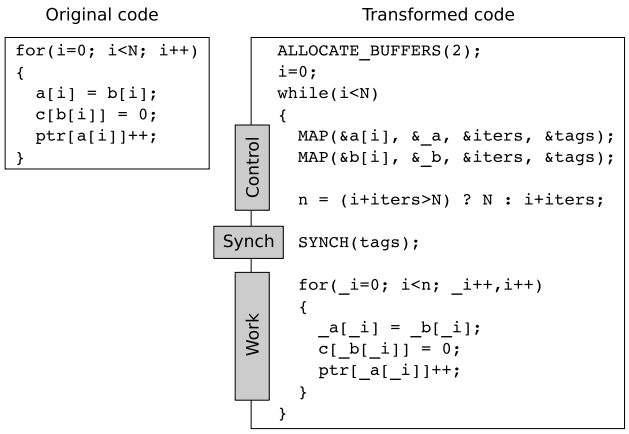

Figure 3: Code transformation for the hybrid memory system.

memory reference that is mapped to the SPM. In Figure 3, `ALLOCATE_BUFFERS` allocates two buffers to map chunks of `a` and `b`, and each buffer occupies half the size of the SPM.

The control phase moves chunks of array sections between the SPM and the GM. At every instance of the control phase, in the `MAP` statement for each array section, the chunk of data for the next work phase is mapped to its corresponding SPM buffer with a *dma-get*, and the previously used chunk is written back to the GM if needed with a *dma-put*. Each call to `MAP` also sets a pointer to the first element of the buffer that has to be computed (`_a` and `_b`), updates the number of iterations that can be performed with the current mappings (`iters`) and sets the tags associated to the DMA transfers (`tags`).

After waiting for the DMA transfers to finish in the synchronization phase, the work phase takes place. This phase does the same computation as the original loop, but with two differences. First, the memory references to the array sections (`a` and `b`) are substituted with their corresponding SPM mappings (`_a` and `_b`). Second, the iteration space of the work phase is limited to the number of iterations that can be performed with the chunks of data currently mapped to the SPM.

2.3. Coherence Problem

The hybrid memory system opens the door to incoherences between copies of data in different coherence domains. When a chunk of data is mapped to some SPM, a copy of the data is created in its address space, and the coherence between the copy in the SPM and the copy in the GM has to be explicitly maintained because there is no hardware coherence between the two memory spaces. This issue restricts the compiler from performing the code transformation in non-trivial cases.

Once the compiler has identified the array sections to be mapped to the SPMs, it changes the memory references in the work phase so that they access the copy in the SPM, while the rest of memory references access the GM. This causes that two incoherent copies of the same data can be accessed simultaneously during the computation, one via strided accesses to the SPM and the other via random accesses to the GM, resulting in an incorrect execution. In order to ensure the

correctness of the code transformation, the compiler has to apply alias analyses [18, 30, 45] between the memory references that target the SPMs and the rest of memory references in the loop body and ensure there is no aliasing. In the example in Figure 3 this implies predicting if any instance of the accesses to `c` or `ptr` aliases with any instance of the accesses to `a` or `b`. This problem, that also affects compiler auto-vectorization and auto-parallelization [24], has not been solved in the general case, especially in the presence of pointers.

A good way to solve the memory aliasing problem for the hybrid memory system is to adopt a lightweight hardware/software co-designed coherence protocol [5, 6] that ensures the compiler can always generate correct and efficient code. The task of the compiler in the coherence protocol is to identify *potentially incoherent accesses*. A potentially incoherent access is a memory access that the compiler cannot ensure it will never access data in the GM that is mapped to some SPM. The compiler marks these accesses and the hardware, at execution time, checks if the data that is being accessed is mapped to some SPM and diverts the access to it.

2.4. Compiler Support for the Coherence Protocol

The compiler support for the coherence protocol consists on identifying memory instructions that may access data in the GM that is mapped to some SPM so that the hardware diverts them to the correct copy of the data. This section gives an overview of how the compiler performs this identification. The details of all the compiler phases and analyses used in this process can be found in [5, 6].

The compiler starts by classifying the memory references of the code based on their access patterns and the possible memory aliasing hazards between them. Then, for the memory references that do not present aliasing problems, it generates memory instructions that will directly access the GM or the SPMs while, for potentially incoherent accesses, the compiler generates *guarded memory instructions* that will be diverted to the appropriate memory at execution time. The categorization of the memory references is:

- *SPM accesses* are those that traverse array sections private to each thread with a strided access pattern. Normal assembly memory instructions are emitted for them, using as source operands a base pointer to a SPM buffer and an offset, so they directly access the SPMs.
- *GM accesses* are those that do not sequentially traverse array sections and do not alias with any SPM access. Normal assembly memory instructions are emitted for them, using as source operands a base pointer to a GM address and an offset, so they directly access the GM.
- *Potentially incoherent accesses* are those that do not sequentially traverse array sections and alias or may alias with some SPM access. *Guarded memory instructions* are emitted for them, using as source operands a base pointer to a GM address and an offset. When it is executed, the guarded memory instruction is identified by the hardware, that checks if the resulting GM address is mapped to some SPM and diverts the access to the corresponding memory. This paper assumes an x86_64 architecture that implements the guard for the memory instructions with an instruction prefix that can be added to any instruction that accesses memory.

In the example in Figure 3 the accesses to `a` and `b` are SPM accesses because they traverse private array sections using a strided access pattern, so they are respectively emitted with a normal store and a normal load. The source operands of the instructions are the pointers to the SPM buffers (`_a` and `_b`) and the offset (`_i`). The computation of the address at execution time will result in an address that is in the virtual address space of the local SPM, so it will be accessed.

Random accesses to `c` and `ptr` are classified as GM or potentially incoherent accesses depending on the outcome of the alias analysis. In the example, the alias analysis succeeds in ensuring that `c` does not alias with any SPM access, so it is categorized as a GM access and a normal store is emitted with `c` as a base address and the content of `_b[_i]` as offset, so the resulting GM address will guide the access to the L1 cache. For `ptr` the example assumes the alias analysis specifies it may alias with some SPM access, so it is classified as a potentially incoherent access. The compiler emits a guarded load, an increment and a guarded store. The base address of these two guarded memory instructions is `ptr` and the offset is the content of `_a[_i]`, which will result in an address in the GM virtual address space. At execution time this GM virtual address will be used to check if the chunk of data is mapped to some SPM, and will be diverted to the appropriate memory.

3. Coherence Protocol

The goal of the hardware support for the coherence protocol is to check if the data accessed by a potentially incoherent access is mapped to some SPM and, in case it is, divert the access to the SPM. With this approach the architecture does not maintain the different copies of the data in a coherent state, but ensures that the valid copy of the data is always accessed.

The proposed hardware design aims to exploit a key characteristic of the applications: it is extremely rare that, in the same loop, the same data is accessed at the same time using strided and random accesses. The data of a program is kept in data structures, and the internal organization of the data structures is what defines the way the data is accessed, thus defines what kind of memory accesses are used to do so. Moreover, although some data structures (e.g., arrays) can be accessed in different ways, it is unnatural to access the same data using strided and random accesses in the same computational parallel loop. This implies that SPM accesses almost never alias with potentially incoherent accesses, although the compiler is unable to ensure it. For this reason, the hardware coherence protocol is designed to not penalize the latency of potentially incoherent accesses that do not access data mapped to the SPMs, which is the most common case.

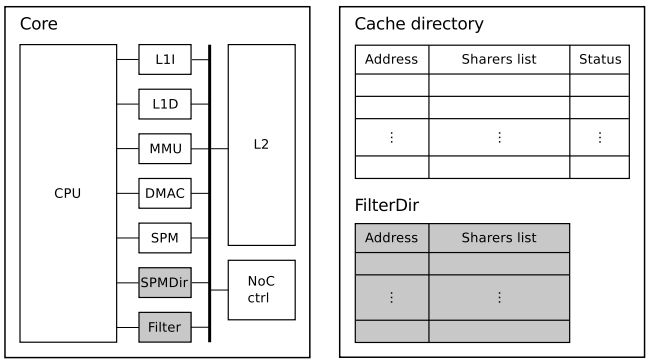

Figure 4: Hardware support for the coherence protocol. Every core is extended with a SPMDir and a Filter, and a FilterDir is added to the cache directory of the cache coherence protocol.

The hardware support for the coherence protocol consists on tracking what data is mapped to the SPMs and what data is known to be not mapped to the SPMs. Figure 4 shows the hardware extensions in one core and in one slice of the directory of the cache coherence protocol. Every core tracks what data is mapped to its SPM in its *SPMDir*, so all the data mapped to all SPMs is tracked in a distributed way. Chunks of data that are not mapped to any SPM and that have been recently accessed by guarded memory instructions are tracked in a hierarchy of filters. Every core has a *filter* that tracks chunks of data not mapped to any SPM and the *filterDir* tracks the contents of all the filters. Placing the filters and the SPMDirs close to the core allows fast access to the information needed to divert guarded accesses.

The next subsections explain the implementation of these hardware structures, how they are operated in the execution of potentially incoherent accesses and how they track what data is mapped to the SPMs and what data is not mapped.

3.1. Implementation of Hardware Structures

The hardware additions are implemented as follows:

- The *SPMDir* is a CAM array that tracks the base GM address of the chunks of data mapped to the SPM of the core.
- The *filter* is a CAM array that keeps base GM addresses not mapped to any SPM.
- The *filterDir* is an extension of the cache directory that consists of a CAM array that keeps base GM addresses and a RAM array with a bitvector of sharer cores that tracks which cores have the address in their filters.

Note that the hardware structures track data at a fixed granularity using 64-bit virtual base addresses. This can be done because, in fork-join parallelism, all threads work with the same SPM buffer size. As explained in Section 2.2, prior to the execution of a loop, the size of the SPM is divided among equally-sized SPM buffers. This buffer size is notified to the hardware, that sets the values of the *Base Mask* and *Offset Mask* internal configuration registers that are used to decompose any address into a base address and an address offset. This allows that, first, base addresses can be used to operate

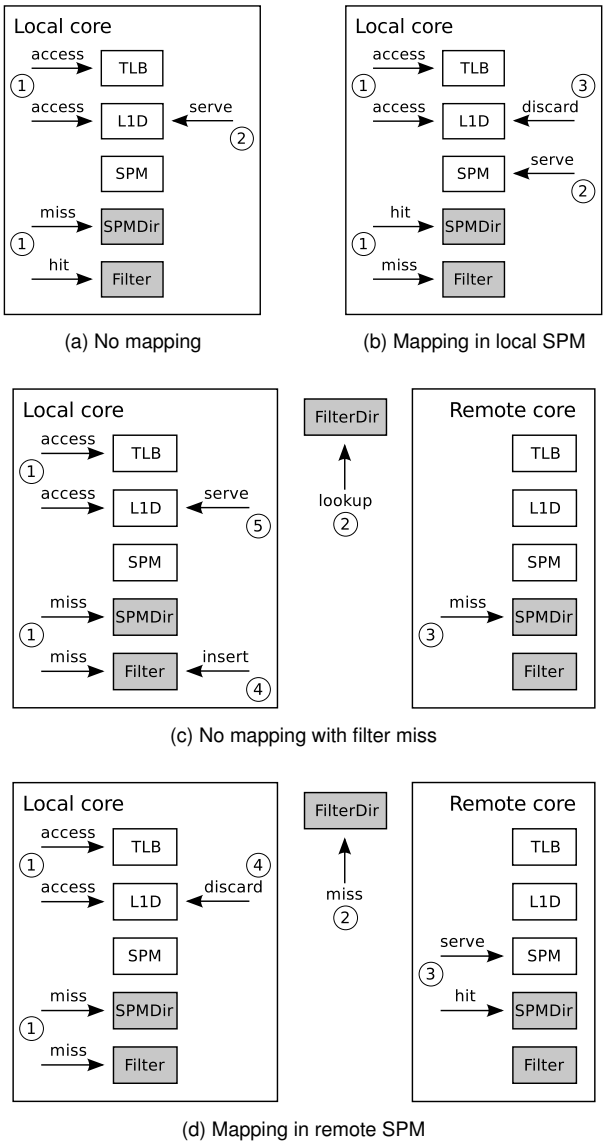

Figure 5: Casuistic of guarded memory accesses.

all the hardware structures and, second, the SPMDir does not need a RAM array to store the SPM base addresses for every entry, since the index of the entry is equivalent to the buffer number and, thus, the base address of the SPM buffer. In other forms of parallelism (e.g, region- or task-based) different threads can simultaneously execute parts of the code that require managing the SPMs at different granularities, so range lookups would be required in the hardware structures.

3.2. Diverting Potentially Incoherent Accesses

The hardware design of the coherence protocol tracks all the data that is mapped to the SPMs in the SPMDirs, and the data that has been recently accessed by potentially incoherent accesses that is not mapped to any SPM is tracked in the hierarchy of filters. The next paragraphs explain how this information is used in the execution of guarded accesses.

When a guarded memory instruction is executed, the TLB and the L1 cache are accessed as in a normal memory instruction for the GM. Additionally, the GM base address is looked up in the filter and the SPMDir. Different situations can arise depending on the results of the lookups, as shown in Figure 5.

In Figure 5a the data accessed by the potentially incoherent access is not mapped to any SPM, so the lookup in the SPMDir misses and the lookup in the filter hits ①. The TLB translates the address and the L1 cache is accessed normally to serve the memory access ②.

In Figure 5b the data is mapped to the SPM of the local core. The lookup in the SPMDir hits ① and the SPM base address corresponding to the entry that returned the hit is added to the offset of the address of the guarded instruction, resulting in the SPM address where the data is mapped. This address is used to access the SPM ②. In addition, if the guarded instruction is a load, the result of the cache access is discarded ③. Note that guarded store instructions always write the data in the L1 cache because, if they alias with a read-only SPM buffer, it is not guaranteed that the software will write-back the SPM buffer to the GM, so the modifications done by guarded store instructions could be lost.

In Figure 5c the data is not mapped to any SPM, though the address is not cached in the filter. The lookup in the SPMDir misses and the lookup in the filter misses ①, so the data may be mapped to a remote SPM. A request is sent to the filterDir and, if the guarded instruction is a load, the L1 cache access is buffered in the MSHR. When the filterDir receives the request it looks up the address ②. If the lookup hits the address is not mapped to any SPM, so the directory responds to the local core. If the lookup misses the request is broadcast to all the remote cores to check if the address is in their SPMDirs ③. The lookup misses in all the cores because the data is not mapped to any SPM, so the remote cores respond to the filterDir and this to the local core. When the local core receives the response the GM base address is inserted in the filter ④ and, if the guarded instruction is a load, the buffered cache access is used to serve the memory access ⑤.

In Figure 5d the data is mapped to the SPM of a remote core. Like in the previous situation, both lookups in the filter and in the SPMDir miss ①, so the L1 cache access is buffered if it is a guarded load and a request is sent to the filterDir, which also misses ② and broadcasts the request to all the remote cores. In this case, one of the remote cores has the data mapped to its SPM, so the lookup in its SPMDir hits ③. The SPM address is calculated using the entry index and the offset of the remote access, and it is used to access the SPM, that serves the remote access. If the remote access is a load, the remote core sends the data to the local core while, if it is a remote store, the remote core writes the data to its SPM and responds with an ACK to the local core. When the response arrives to the local core, if it is a guarded load, the buffered cache access is discarded ④ and the data from the remote core is forwarded to the CPU.

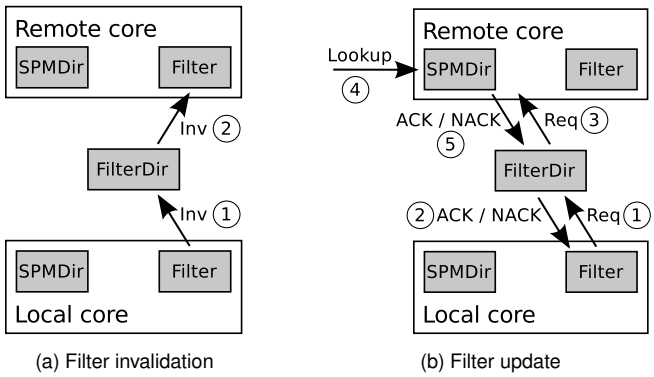

Figure 6: Filter invalidation and update.

3.3. Tracking SPMs Contents

The contents of the hardware structures of the coherence protocol are used to divert the guarded accesses. The next paragraphs explain how all the information of the contents of the SPMs is tracked in the SPMDirs, the filters and the filterDir.

When a core maps data to its SPM, its SPMDir is updated and the filters are invalidated. The source GM address and the destination SPM address of the *dma-get* are used to calculate the GM base address and the SPM base address, and the SPMDir entry associated to the SPM base address is updated with the GM base address. Figure 6a shows how the filters are invalidated. First, an invalidation message is sent for the GM base address to the filterDir ①, which triggers a lookup of the address. If the address is not found no core has the address in its filter, so no more actions take place. If the GM base address is in the filterDir, a filter invalidation message is sent to all the cores in the sharers list ②. When the sharer cores receive the invalidation message they look up the address in their filter and invalidate the entry.

The filters and the filterDir are updated when the execution of a potentially incoherent access triggers a lookup of a GM base address in a filter and misses (Figures 5c and 5d). The mechanism to update the filters and the filterDir is shown in Figure 6b, and is very similar to how caches operate on a cache miss. When a lookup of a GM base address misses in a filter, it has to check if the GM base address is mapped to some SPM. First, a request for the GM base address is sent to the filterDir ①, which looks up the address. If the address is in the filterDir it means it is not mapped to any SPM, so the requestor core is added to the sharers list and an ACK is sent as a response ②. When the requestor core receives the ACK it inserts the GM base address in its filter. If the lookup in the filterDir after ① misses, the GM base address may be mapped to some SPM. The filterDir broadcasts a request of the address to all the cores ③. When the cores receive the request, they lookup the address in their SPMDir ④ and respond to the filterDir with an ACK or a NACK depending if the lookup hit or missed ⑤. If all the cores respond NACK, the data is not mapped to any SPM, so the filterDir inserts the GM base

address, sets the local core in the sharers list and sends it a response with ACK ②. When the local core receives the ACK it inserts the GM base address in its filter. If a remote core responds to the broadcast request with an ACK ⑤ it means it has the data mapped to its SPM, so the address cannot be filtered. The filterDir responds with a NACK to the local core and this does not update its filter.

Note that the filter of every core and the filterDir are associative buffers with a replacement policy so, when they are updated, an older entry can be evicted. When an entry of the filter of a core is evicted, a message is sent to the filterDir to notify the eviction, and this removes the core from the sharers list. When the filterDir is updated and an older entry is evicted, an invalidation message is sent to all the sharer cores, which invalidate their filter, like in step ② of Figure 6a.

3.4. Memory Consistency Model

Another important problem that affects the ability of the compiler to generate code for the SPMs of the hybrid memory system is the memory consistency model provided by the architecture. The memory consistency model defines what is the expected behaviour of a sequence of memory operations. The vast majority of shared memory programming models rely on sequential consistency models [3] that ensure that a sequence of memory operations will happen in program order inside the same thread, while a sequence of memory operations from different threads can happen in any order. For this reason, the programmer has to use synchronization mechanisms to avoid data races between the threads, but not between the memory accesses of each thread. In order to allow the compiler to transform the code for the hybrid memory system, the memory subsystem has to ensure these conditions are fulfilled.

The coherence protocol for the hybrid memory system breaks the sequential consistency rules between potentially incoherent accesses and SPM accesses of the same thread. The virtual address of potentially incoherent accesses is initially a GM address that is changed to a different SPM virtual address if the data is mapped to the SPM. This causes that, when a thread accesses the same data using strided and potentially incoherent accesses, an out-of-order core can re-order the instructions and the Load/Store Queue (LSQ) will not detect an ordering violation has happened because the virtual addresses of the two accesses are different. When the potentially incoherent access is sent to memory and its address is changed to point to the SPM it results in the same address of the strided access so, if at least one of the two accesses was a store, the re-ordering breaks the sequential consistency rules. In order to solve this problem, when a potentially incoherent access hits in the SPMDir, the new SPM address is notified to the LSQ to re-check the ordering for the new address, and the pipeline is flushed if a violation is found. Note that if the guarded access aliases with the contents of a remote SPM it is responsibility of the programmer to synchronize the accesses between the threads, just like it is done for any other type of accesses.

4. Summary and Discussion

The architecture and the system organization proposed in this paper achieve that transparently managed SPMs can be added to the memory hierarchy of shared memory manycore architectures. The compiler categorizes the memory accesses of a conventional parallel loop. For strided accesses to private array sections, the compiler transforms the code so that a runtime library maps the data to the SPMs. Random accesses are served by the cache hierarchy if the compiler can ensure they do not alias with the contents of some SPM. When the compiler is unsure of the aliasing of a random access, it generates a guarded memory instruction. At execution time, the hardware support of the coherence protocol decides if these accesses are served by some SPM or by the cache hierarchy.

Although the hardware support of the coherence protocol resembles the operation of a directory-based cache coherence protocol, it does not undermine the benefits of the hybrid memory system compared to a cache hierarchy. The wide majority of memory accesses in HPC applications are strided accesses to array sections, and these are served by the SPMs in a very energy efficient way, without any CAM lookup in the TLB nor in the tags of the caches nor in any hardware structure of the coherence protocol. In addition, SPM accesses do not suffer performance penalties due to cache misses, and the coherence traffic for them is reduced because accesses to the SPMs do not generate coherence traffic and the DMA transfers allow to move data in a very efficient way. Random accesses to the cache behave like in a conventional cache coherent system, without any overhead. Potentially incoherent accesses can suffer performance penalties if the filter misses, and are always a bit less power efficient due to the lookups in the filter and in the SPMDir. Since these accesses are a minority, the number of CAM lookups and the coherence traffic added by the filters are overweighted by the CAM lookups and the coherence traffic saved for the SPM accesses, preserving the benefits of the hybrid memory system.

4.1. OS Support and Backwards Compatibility

The hybrid memory system can be made backwards compatible with minor operating system (OS) changes and without impacting performance. The OS process structure is extended with the fields necessary to manage the registers for the virtual-to-physical address mapping of the SPMs at context switch. By default, processes execute with this mapping disabled, so the SPMs are not accessible. Whenever a SPM-enabled application starts, the OS configures the registers for the address spaces of the SPMs and stores their values in the process structure. Whenever a process is scheduled, the registers are set to the values stored in the process structure.

Since multiple concurrent applications can be using the SPMs, the OS must also save and restore the contents of the SPMs whenever a process is scheduled for execution. In order to keep these overheads low, SPM contents can be switched

Table 1: Main simulator parameters.

Cores	64 cores, Out-of-order, 6 instructions wide, 2GHz
Pipeline front end	13 cycles. Branch predictor 4K selector, 4K G-share, 4K Bimodal. 4-way BTB 4K entries. RAS 32 entries
Execution	ROB 160 entries. IQ 64 entries. LQ/SQ 48/32 entries. 3 INT ALU, 3 FP ALU, 3 Ld/St units. 256/256 INT/FP RegFile. Full bypass.
L1 I-cache	2 cycles, 32 KB, 4-way, pseudoLRU
L1 D-cache	2 cycles, 32 KB, 4-way, pseudoLRU, stride prefetcher
L2 cache	Shared unified NUCA 16MB, sliced 256KB/core 15 cycles, 16-way, pseudoLRU
Cache coherence	Real MOESI with blocking states, 64B line size distributed 4-way cache directory 64K entries
NoC	Mesh, link 1 cycle, router 1 cycle
SPM	2 cycles, 32 KB, 64B blocks
DMAC	DMA command queue 32 entries, in-order Bus request queue 512 entries, in-order
SPMDir	32 entries
Filter	48 entries, fully associative, pseudoLRU
FilterDir	Distributed 4K entries, fully associative, pseudoLRU

lazily, similar to how the Linux kernel does for the floating point register file [1]. Additionally, a register that contains a single bit for each SPM in the system is added to each core. The Nth bit of this register identifies whether the Nth SPM can be accessed, and accessing a SPM without that bit set raises an exception. This mechanism allows the OS to accurately control which SPMs are available to the running processes, making possible that multiple processes that use the SPMs can be safely executed concurrently.

These OS modifications provide backwards compatibility and can also be used by the OS to improve energy consumption by powering down the SPMs that are not being actively used. As a result, the only overhead when running in compatibility mode is the unused area of the SPMs, the DMACs and the hardware structures of the coherence protocol.

5. Evaluation

This section evaluates the overheads of the proposed coherence protocol and then compares the resulting hybrid memory system with a cache-based system.

5.1. Experimental Framework

The proposal has been evaluated with Gem5 [12], using its x86 cycle-accurate detailed out-of-order core model and its detailed memory hierarchy model. The energy consumption has been evaluated with McPAT [32] using a process technology of 22nm and the default clock gating scheme of the tool. Both simulators are extended with the SPMs, the DMACs and the hardware support for the coherence protocol. Table 1 shows the main configuration parameters of the simulators.

Six representative memory intensive HPC benchmarks from the NAS benchmark suite [8] are used for the evaluation. They have been compiled using GCC 4.7.3 with the -O3 optimization flag on. The code transformations for the hybrid memory system have been done manually and the alias analysis report of GCC has been used to identify the potentially incoherent accesses. For the memory references that GCC is not able to determine the aliasing, an assembly macro is added to the source code to add the x86 instruction prefix that implements the guard for potentially incoherent accesses.

Table 2: Benchmarks and memory access characterization.

Benchmark			SPM refs		Guarded refs	
Name	Input	Kernels	#	Data Size	#	Data Size
CG	ClassB	1	5	109 MB	1	600 KB
EP	ClassA	2	3	1 MB	1	512 KB
FT	ClassA	5	32	269 MB	4	1 MB
IS	ClassA	1	3	67 MB	2	2 MB
MG	ClassA	3	59	454 MB	6	64 B
SP	ClassA	54	497	2 MB	0	0 B

The runtime library that manages the SPMs is a modified version of a software cache for the Cell B.E. [22] that has been ported to the hybrid memory system and optimized. Double buffering is not used.

5.2. Benchmark Characterization

Table 2 shows the main characteristics of the benchmarks. For each benchmark it is shown the input class, the number of kernels, the number of strided and potentially incoherent memory references and the sizes of the data sets accessed by each type of accesses. These benchmarks allow to study many different scenarios regarding the number of memory accesses of each type and the sizes of the data sets they access. CG and IS use few strided references to traverse big input sets and few guarded references to access a smaller data set, so the ratio of guarded accesses with respect to the total number of accesses is high. In EP the amount of references of each type is low and the data sets are small and, in addition, its kernels use many scalar temporal values that cause register spilling, so the majority of memory accesses are to the stack. FT and MG traverse big input sets with many strided references and use a few guarded references to access much smaller parts of the data set, while SP traverses a small input set in a series of 54 different kernels that only use SPM accesses.

Like in real HPC applications [36, 44], the characteristics of the benchmarks fulfill the motivation for the introduction of SPMs. In all benchmarks the number of strided memory references is higher than the number of guarded references, and the data set accessed by SPM accesses is much bigger than the one accessed by guarded accesses. Furthermore, the data sets accessed by SPM and guarded accesses are disjoint, though the compiler is unable to ensure it using alias analyses.

5.3. Coherence Protocol Overheads

This section evaluates the overheads of the coherence protocol in a 64-core manycore. In the study, the proposed coherence protocol is compared with an ideal coherence protocol that diverts guarded accesses to the correct copy of the data without the need of SPMDirs, filters, the filterDir nor any traffic to maintain them. Figure 7 shows the overheads in performance, energy and network traffic of the hybrid memory system aug-

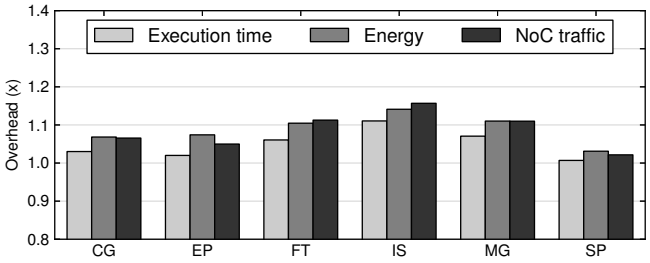

Figure 7: Overhead in execution time, energy consumption and NoC traffic added by the coherence protocol.

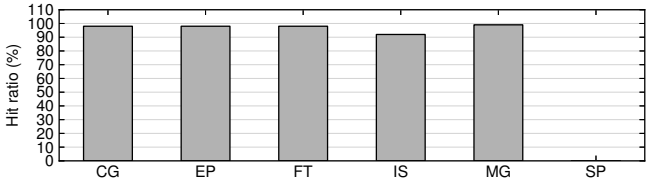

Figure 8: Filter hit ratio.

mented with the proposed coherence protocol with respect to the hybrid memory system with ideal coherence.

The coherence protocol adds performance overheads of 1% to 11%. These overheads are caused by the increase in network traffic and by filter misses at the execution of guarded accesses. The filter hit ratio for each benchmark is shown in Figure 8. In SP the performance overhead is negligible because no guarded accesses are generated and very few DMA transfers are issued, so the filters are not used and the overhead in network traffic is only 2%. CG, EP, FT and MG present performance overheads between 2% and 7%. Although the filter hit ratio is at least 97%, the network traffic added by the coherence protocol, between 4% and 11%, slightly penalizes performance. In IS the performance overhead is 11% because the filter hit ratio is lower, 92%, which penalizes the latency of guarded accesses and adds higher overheads in network traffic, 15%. In all benchmarks the contention in the filterDir is very low due to the low rate of DMA transfers and filter misses. In addition, guarded accesses never alias with SPM accesses and the filterDir can track the whole data set accessed by guarded accesses, so filter invalidations and pipeline squashes due to ordering violations never happen.

In terms of energy consumption, the overheads range from 3% to 14%. In SP the overhead is only 3% because the filters are gated off and only the SPMDirs and the filterDir consume some extra energy at every DMA transfer. In the rest of benchmarks the overheads range between 7% and 14%. Almost half of the overhead is due to static energy consumption, where SPMDirs, filters and the filterDir add overheads of less than 3% and the performance overheads add another 2% to 5% in the whole chip. In dynamic energy the SPMDirs add up to 3% overhead, while less than 2% is added by the filters and the filterDir. The extra traffic causes overheads of less than 1% in the NoC in all cases except in IS, where it adds 3%.

In terms of area, the SPMDirs, the filters and the filterDir add an overhead of less than 4%.

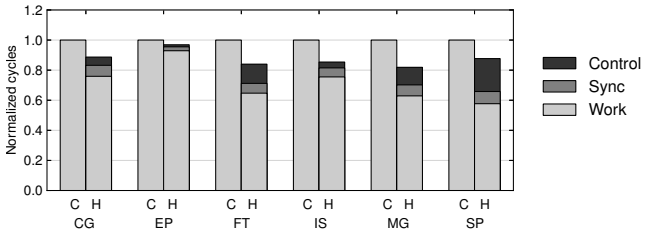

Figure 9: Performance of the cache-based system (C) and the hybrid memory system (H).

In conclusion, the proposed coherence protocol introduces very low overheads in performance, energy consumption and network traffic. The overheads are caused by the extra traffic added at DMA transfers, and also due to filter misses in IS. The average overheads in performance, energy consumption and network traffic are 4%, 9% and 8%, respectively.

5.4. Comparison with Cache-Based Systems

The immediate result of the coherence protocol is that any computational loop can be executed on the hybrid memory system without any programming effort. In order to demonstrate the benefits of this accomplishment, in this section the hybrid memory system is compared with a cache-based system in terms of performance, energy consumption and network traffic in a 64-core manycore. The studied hybrid memory and cache-based systems have the same characteristics, shown in Table 1, but with one difference. For fairness, the L1 D-cache of the cache-based system is augmented to 64KB without affecting access latency, matching the 32KB L1 D-cache plus the 32KB SPM of the hybrid memory system.

Figure 9 shows the performance of the two systems. Both bars are normalized to the cache-based system execution time and show the time spent in each execution phase. The figure shows that the hybrid memory system reduces the execution time with respect to the cache-based system for all benchmarks, achieving execution time reductions between 3% and 18% (or speedups of 1.03x to 1.22x). EP is dominated by cache accesses to the stack and a very small data set is mapped to the SPMs, so both systems have practically the same behaviour and the performance differences are negligible. In the rest of benchmarks speedups of 1.12x to 1.22x are achieved due to the ability of the hybrid memory system to serve memory accesses more efficiently than the cache-based system. In FT, MG and SP, which have a lot of strided accesses, the hybrid memory system always serves these accesses with the SPMs without performance penalties while, in the cache-based system, it is observed that cache misses happen because the prefetchers are not able to provide all the data required by all the strided references on time, and also the big amount of data brought and prefetched to the L1 caches causes conflict misses. In benchmarks that use guarded accesses, especially IS and CG but also FT and MG, performance benefits come from exploiting their temporal locality. The hybrid memory system serves most of the memory accesses with the SPMs so

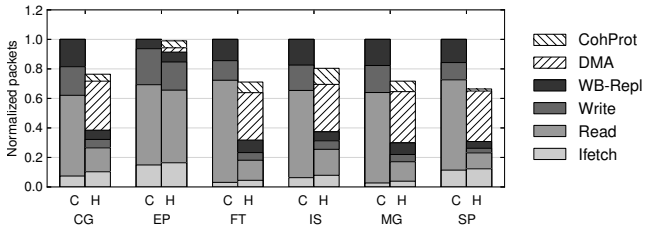

Figure 10: NoC traffic of the cache-based system (C) and the hybrid memory system (H).

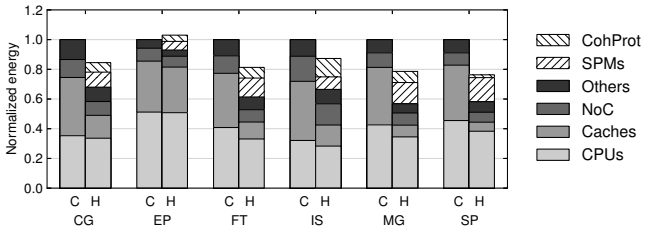

Figure 11: Energy consumption of the cache-based system (C) and the hybrid memory system (H).

the data brought to the L1 cache is much less often evicted, while in the cache-based system it is quickly evicted to bring and prefetch data for the strided accesses. In addition, Figure 8 shows that the filters of the coherence protocol also present very high hit ratios, more than 92% in all cases, so the latency of guarded memory accesses is not penalized neither. These factors allow the hybrid memory system to serve data more efficiently than the cache-based system, reducing the execution time of the work phase by 25% to 43%. These speedups in the work phase overweight the overheads added in the control and synchronization phases in all cases, resulting on an average speedup of 1.14x across all benchmarks.

The NoC traffic of the cache-based and the hybrid memory systems is shown in Figure 10. The two bars are normalized to the traffic in the cache-based system and categorize the type of traffic in groups: instruction fetch requests (Ifetch), data cache reads (Read), data cache writes (Write), which include data requests, prefetch requests, data and acknowledgment packets; write-back and replacements of cache lines (WB-Repl), which includes write-back requests, replacements, invalidations, data and acknowledgment packets; DMA transfers (DMA), which includes DMA requests, data and acknowledgment packets; and coherence traffic introduced by the coherence protocol (CohProt), which includes all the traffic explained in Section 3. Results show that the hybrid memory system reduces the network traffic by 20% to 34% in all benchmarks except EP. In these benchmarks traffic is reduced because most of the data set is mapped to the SPMs and moved using DMA transfers, so many cache accesses, misses and prefetches and their associated NoC traffic are eliminated. On the one hand, this eliminates between 71% and 83% of the traffic due to data cache reads, between 61% and 74% of the traffic due to data cache writes, and between 41% and 71% of the traffic in the WB-Repl group. On the other hand, the DMA transfers required to move the data set to the SPMs add 32% to 37% to the total network traffic. In addition, the code transformations for the hybrid memory system and the code of the runtime library cause a higher number of instruction fetches, adding an extra overhead of up to 3%. The proposed coherence protocol also adds NoC traffic, 1% in SP where there are no guarded accesses, 4% to 7% in CG, FT and MG and up to 10% in IS where traffic is introduced at DMA transfers and at filter misses. EP maps a very small data set to the SPMs and does an intensive utilization of the cache hierarchy, so the small traffic reductions achieved by mapping data to the SPMs are compensated with the small overhead added by the coherence protocol and the extra code. On average, the hybrid memory system reduces the network traffic by 29%.

Another benefit of the hybrid memory system is that it consumes less energy than a cache-based system, as shown in Figure 11. The bars show the energy consumption of both systems, normalized to the one of the cache-based system, and they also show how the consumed energy is distributed between the CPUs, the caches (including MSHRs and prefetchers), the NoC, other components such as the cache coherence protocol structures, DMACs and memory controllers (Others), the SPMs, and the structures of the proposed coherence protocol (CohProt). CG, FT, IS, MG and SP present energy savings of 13% to 24%. Important energy savings come from the cache hierarchy, which contributes with more than 35% of the total energy consumed in the cache-based system and is reduced by a factor of 2.5x to 6.1x in the hybrid memory system. This is because most memory accesses are served by the SPMs, which consume between 12% and 16% of the total energy to do so. The hardware structures of the proposed coherence protocol consume between 6% and 12% of the total energy except in SP, where it represents only 1% because no guarded accesses are used. The energy consumed in the CPUs is reduced between 5% and 23%, depending on how many instruction re-executions due to cache misses are avoided, while the reduction in NoC traffic also reduces by 18% to 42% the energy consumed in this component. In EP the SPMs of the hybrid memory system are underutilized and the static energy of the added structures causes an overhead of 3%. The energy saved in all benchmarks is, on average, 17%.

In conclusion, the ability of the hybrid memory system to serve memory accesses with the SPMs and to efficiently move the data with DMA transfers provides important benefits when compared to a cache-based systems. An average speedup of 1.14x is achieved due to the efficiency on serving memory accesses without performance penalties. In addition, using DMA transfers leads to an average reduction in network traffic of 29%, and serving most of the memory accesses with the SPMs instead of the L1 caches is the biggest factor to reduce the energy consumption by 17%, on average.

6. Related Work

6.1. SPM Management in Hybrid Memory Systems

Several forms of hybrid memory systems have been proposed in the past. Although the architecture details of every solution are similar, the way of managing the SPMs changes from one proposal to another. Two big families of approaches can be used: static or dynamic mappings.

Static mapping schemes allocate data in the SPMs at the beginning of the execution and the contents of the SPMs do not change during the computation. This model is used in the embedded domain, where the compiler identifies data to be mapped to the SPMs [7, 38, 42], and in GPUs, where the programmer uses keywords [2] to declare what data is allocated in the SPMs. On the one hand, static mappings do not allow to map big amounts of data to the SPMs due to their limited size. The dynamic approach is better suited for HPC applications because it allows to map big inputs sets to the SPMs in chunks, thus serving most memory accesses with the SPMs and maximizing the benefits. On the other hand, static methods do not encounter the coherence problem of dynamic mappings because data is not replicated in the two storages.

Some works propose hybrid memory systems where the SPMs are managed with a dynamic approach like the one assumed in this paper (explained in Section 2.2). Bertran et al. [11] propose to add a SPM alongside the L1 cache of a single core processor and give the compiler the responsibility of doing the code transformations, but they do not solve the coherence problem between the two storages. Instead, the compiler discards the code transformations in case of unknown memory aliasings, restricting the effective utilization of the SPM. The authors of Virtual Local Stores [15] (VLSs) allow to configure parts of the caches as virtual SPMs, and they offer the programmer a high-level API to dynamically map data to the VLSs. The coherence protocol proposed in this paper could be adopted by these two works to allow the compiler to always generate code to manage the SPMs, improving the programmability of the resulting systems.

Alvarez et al. [5, 6] propose a hardware/software co-designed coherence protocol to add a SPM alongside the L1 cache of a single core. Although the compiler support for the coherence protocol proposed in [5, 6] is exactly the same as the one assumed in this paper, in the rest of aspects there are important differences. At the architecture level, a key difference is that this paper allows every core to access any SPM, while in [5, 6] they discuss that, in a multicore, every core would only have access to its SPM. Regarding the hardware design for the coherence protocol, in [5, 6] every core tracks the data mapped to its SPM independently, but there is no way to make this information globally visible to the other cores, so the execution of a potentially incoherent access cannot check if the data is mapped to the SPM of a remote core. These differences make the coherence protocol in [5, 6] not applicable to manycore architectures. If the SPMs and the SPMDirs are not globally visible, the hardware cannot divert potentially incoherent accesses to remote cores. Instead, the compiler has to wrap the potentially incoherent accesses with a piece of code that does a software lookup to check if some SPM has a copy of the data, triggers a fine-grained DMA transfer to bring the data to the SPM of the local core, accesses it, and triggers a DMA transfer back if the data is modified. This solution adds huge overheads, as observed in previous works in the area of software caches [22]. This paper avoids these overheads by allowing all the cores to access all the SPMs and by proposing a hierarchy of directories and filters that efficiently tracks the contents of all SPMs and diverts the potentially incoherent accesses to any SPM of the chip. Moreover, this paper evaluates a hybrid memory manycore and solves two important problems for the adoption of the hybrid memory system in manycore architectures: the sequential consistency rules in the memory model and the backwards compatibility.

Some architectures [25, 33, 40] offer the possibility to configure the memory hierarchy as a combination of SPMs and caches. These works focus on the hardware details that allow the configurability, but do not discuss how the resulting configuration would be exposed to the upper layers of the system. A promising solution to hide the complexity of the these hybrid reconfigurable memory hierarchies to the programmer is to manage them using compiler and runtime system techniques in a hardware/software co-designed fashion [3, 43]. In this direction, the software layers and the coherence protocol proposed in this paper could be adopted by these architectures to transparently reconfigure and manage the memory hierarchy.

6.2. Coherence Domains

Some works propose to deactivate coherence in cache hierarchies to eliminate the unnecessary coherence traffic. The technique consists on identifying private and shared data and deactivating the cache coherence protocol for the private data.

Cuesta et al. [16] propose to deactivate cache coherence at the granularity of virtual memory pages, using OS support at TLB misses to identify private pages and tracking the privateness with a bit in the page table and in the TLB. Very similar schemes are used to deactivate coherence for shared read-only pages [17], to filter snoop requests [28] and to optimize data placement in NUCA caches [23].

Other authors propose to deactivate cache coherence at the granularity of regions of arbitrary size, tracking them at the microarchitecture level with hardware support. Cantin et al. [13] propose to identify and track private regions in a hardware structure that directly interacts with an ad-hoc coherence protocol. RegionScout [34] identifies private and shared memory regions and uses additional hardware support to filter broadcasts in snoop-based cache coherence protocols. Alisafaee [4] identifies regions where coherence is not needed and introduces the temporal dimension in the region classification, allowing to transition from shared to private when two cores do not access a shared region at the same time.

Some works use the information provided by the programmer or the programming model. Cohesion [27] offers an API to the programmer to dynamically register memory regions in hardware- or software-managed coherence domains, and also proposes a mechanism to allow the transition between domains. With different motivations, DeNovo [14] exploits the data-race-freedom of disciplined programming models to eliminate the transient states of cache coherence protocols.

In general, these works aim to decrease the coherence traffic and the size of the directory of the coherence protocol. In the hybrid memory system, mapping data to the SPMs can be considered a form of deactivating the cache coherence protocol, but in this case the goal is not only to minimize the cache coherence protocol overheads but also to reduce the energy consumed by memory accesses. The code transformations required to map data to the SPMs forces the identification of private data to happen at compile time, and the proposed coherence protocol is required to do it in non-trivial cases.

7. Conclusions

The increasing number of cores in manycore architectures causes severe power consumption and scalability problems in the cache hierarchy. A good solution is introducing SPMs in the memory subsystem but, for the success of this solution, it is key to hide from the programmer the programmability difficulties inherent to SPMs. Although compilers can generate code for the SPMs in trivial cases, they are unable to do so in the presence of pointers with unknown memory aliases.

This paper proposes a hardware coherence protocol that allows the compiler to mark potentially incoherent memory accesses, leaving the responsibility to the hardware of diverting them to the correct copy of the data. The coherence protocol consists of a set of directories and filters that track what data is mapped to the SPMs and what data is not mapped to any SPM. This information is used to divert the potentially incoherent accesses to the SPMs or to the cache hierarchy, ensuring the valid copy of the data is always accessed.

With the coherence protocol the compiler can always generate code for the hybrid memory system, so the architecture can be exposed to the programmer as a shared memory manycore with a sequential consistency memory model that is fully backwards compatible. In a 64-core manycore, compared to a cache hierarchy, the hybrid memory system achieves an average speedup of 1.14x and average reductions in NoC traffic and energy consumption of 29% and 17%, respectively.

Acknowledgments

This work has been supported by the Spanish Government (grant SEV-2011-00067 of the Severo Ochoa Program), by the Spanish Ministry of Science and Innovation (contract TIN2012-34557), by Generalitat de Catalunya (contracts 2014-SGR-1051 and 2014-SGR-1272), and by the RoMoL ERC Advanced Grant (GA 321253). Miquel Moreto has been partially supported by the Ministry of Economy and Competitiveness under Juan de la Cierva postdoctoral fellowship number JCI-2012-15047, and Marc Casas is supported by the Secretary for Universities and Research of the Ministry of Economy and Knowledge of the Government of Catalonia and the Cofund programme of the Marie Curie Actions of the 7th R&D Framework Programme of the European Union (Contract 2013 BP_B 00243).

References

[1] *Intel 64 and IA-32 Architectures Software Developer's Manual*, 2011.

[2] *NVIDIA CUDA C Programming Guide. Version 4.2*, 2012.

[3] S. V. Adve and H.-J. Boehm, "Memory Models: A Case for Rethinking Parallel Languages and Hardware," *Communications of the ACM*, vol. 53, no. 8, pp. 90–101, 2010.

[4] M. Alisafaee, "Spatiotemporal Coherence Tracking," in *MICRO 45: Proceedings of the 45th International Symposium on Microarchitecture*. IEEE Computer Society, 2012, pp. 341–350.

[5] L. Alvarez, L. Vilanova, M. Gonzàlez, X. Martorell, N. Navarro, and E. Ayguadé, "Hardware-Software Coherence Protocol for the Coexistence of Caches and Local Memories," in *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society, 2012, pp. 89:1–89:11.

[6] L. Alvarez, L. Vilanova, M. Gonzàlez, X. Martorell, N. Navarro, and E. Ayguadé, "Hardware-Software Coherence Protocol for the Coexistence of Caches and Local Memories," *IEEE Transactions on Computers*, vol. 64, no. 1, pp. 152–165, 2015.

[7] O. Avissar, R. Barua, and D. Stewart, "An Optimal Memory Allocation Scheme for Scratch-pad-based Embedded Systems," *ACM Transactions on Embedded Computing Systems*, vol. 1, no. 1, pp. 6–26, 2002.

[8] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The NAS Parallel Benchmarks," in *SC '91: Proceedings of the 1991 Conference on Supercomputing*. IEEE Computer Society, 1991, pp. 158–165.

[9] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel, "Scratchpad Memory: A Design Alternative for Cache On-chip Memory in Embedded Systems," in *CODES '02: Proceedings of the 10th International Symposium on Hardware/Software Codesign*. ACM, 2002, pp. 73–78.

[10] T. B. Berg, "Maintaining I/O Data Coherence in Embedded Multicore Systems," *IEEE Micro*, vol. 29, no. 3, pp. 10–19, 2009.

[11] R. Bertran, M. Gonzàlez, X. Martorell, N. Navarro, and E. Ayguadé, "Local Memory Design Space Exploration for High-Performance Computing," *The Computer Journal*, vol. 54, no. 5, pp. 786–799, 2010.

[12] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 Simulator," *SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[13] J. F. Cantin, M. H. Lipasti, and J. E. Smith, "Improving Multiprocessor Performance with Coarse-Grain Coherence Tracking," in *ISCA '05: Proceedings of the 32nd International Symposium on Computer Architecture*. ACM, 2005, pp. 246–257.

[14] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C.-T. Chou, "DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism," in *PACT '11: Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*. IEEE Computer Society, 2011, pp. 155–166.

[15] H. Cook, K. Asanovic, and D. A. Patterson, "Virtual Local Stores: Enabling Software-Managed Memory Hierarchies in Mainstream Computing Environments," Electrical Engineering and Computer Sciences Department, University of California at Berkeley, Tech. Rep. UCB/EECS-2009-131, 2009.

[16] B. Cuesta, A. Ros, M. E. Gómez, A. Robles, and J. Duato, "Increasing the Effectiveness of Directory Caches by Deactivating Coherence for Private Memory Blocks," in *ISCA' 11: Proceedings of the 38th International Symposium on Computer Architecture*. ACM, 2011, pp. 93–104.

[17] B. Cuesta, A. Ros, M. E. Gómez, A. Robles, and J. Duato, "Increasing the Effectiveness of Directory Caches by Avoiding the Tracking of Noncoherent Memory Blocks," *IEEE Transactions on Computers*, vol. 62, no. 3, pp. 482–495, 2013.

[18] A. Deutsch, "Interprocedural May-Alias Analysis for Pointers: Beyond k-limiting," in *PLDI '94: Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*. ACM, 1994, pp. 230–241.

[19] A. E. Eichenberger, J. K. O'Brien, K. M. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, M. K. Gschwind, R. Archambault, Y. Gao, and R. Koo, "Using Advanced Compiler Technology to Exploit the Performance of the Cell Broadband Engine™ Architecture," *IBM Systems Journal*, vol. 45, no. 1, pp. 59–84, 2006.

[20] A. E. Eichenberger, K. O'Brien, K. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, and M. Gschwind, "Optimizing Compiler for the CELL Processor," in *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*. ACM, 2005, pp. 161–172.

[21] P. N. Glaskowsky, *NVIDIA's Fermi: The First Complete GPU Computing Architecture*, 2009, white paper.

[22] M. Gonzàlez, N. Vujic, X. Martorell, E. Ayguadé, A. E. Eichenberger, T. Chen, Z. Sura, T. Zhang, K. O'Brien, and K. O'Brien, "Hybrid Access-Specific Software Cache Techniques for the Cell BE Architecture," in *PACT '08: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*. ACM, 2008, pp. 292–302.

[23] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches," in *ISCA '09: Proceedings of the 36th International Symposium on Computer Architecture*. ACM, 2009, pp. 184–195.

[24] L. J. Hendren, J. Hummell, and A. Nicolau, "Abstractions for Recursive Pointer Data Structures: Improving the Analysis and Transformation of Imperative Programs," in *PLDI '92: Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*. ACM, 1992, pp. 249–260.

[25] N. Jayasena, "Memory Hierarchy Design for Stream Computing," Ph.D. dissertation, Stanford University, 2005.

[26] J. Kahle, "The Cell Processor Architecture," in *MICRO 38: Proceedings of the 38th International Symposium on Microarchitecture*. IEEE Computer Society, 2005, pp. 3–4.

[27] J. H. Kelm, D. R. Johnson, W. Tuohy, S. S. Lumetta, and S. J. Patel, "Cohesion: An Adaptive Hybrid Memory Model for Accelerators," *IEEE Micro*, vol. 31, no. 1, pp. 42–55, 2011.

[28] D. Kim, J. Ahn, J. Kim, and J. Huh, "Subspace Snooping: Filtering Snoops with Operating System Support," in *PACT '10: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*. ACM, 2010, pp. 111–122.

[29] M. Kistler, M. Perrone, and F. Petrini, "Cell Multiprocessor Communication Network: Built for Speed," *IEEE Micro*, vol. 26, no. 3, pp. 10–23, 2006.

[30] W. Landi and B. G. Ryder, "A Safe Approximate Algorithm for Interprocedural Aliasing," in *PLDI '92: Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*. ACM, 1992, pp. 473–489.

[31] J. Leverich, H. Arakida, A. Solomatnikov, A. Firoozshahian, M. Horowitz, and C. Kozyrakis, "Comparing Memory Systems for Chip Multiprocessors," in *ISCA '07: Proceedings of the 34th Annual International Symposium on Computer Architecture*. ACM, 2007, pp. 358–368.

[32] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures," in *MICRO 42: Proceedings of the 42nd International Symposium on Microarchitecture*. IEEE Computer Society, 2009, pp. 469–480.

[33] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz, "Smart Memories: A Modular Reconfigurable Architecture," in *ISCA '00: Proceedings of the 27th International Symposium on Computer Architecture*. ACM, 2000, pp. 161–171.

[34] A. Moshovos, "RegionScout: Exploiting Coarse Grain Sharing in Snoop-Based Coherence," in *ISCA '05: Proceedings of the 32nd International Symposium on Computer Architecture*. ACM, 2005, pp. 234–245.

[35] R. Murphy, "On the Effects of Memory Latency and Bandwidth on Supercomputer Application Performance," in *IISWC '07: Proceedings of the 10th International Symposium on Workload Characterization*. IEEE Computer Society, 2007, pp. 35–43.

[36] R. C. Murphy and P. M. Kogge, "On the Memory Access Patterns of Supercomputer Applications: Benchmark Selection and Its Implications," *IEEE Transactions on Computers*, vol. 56, no. 7, pp. 937–945, 2007.

[37] Y. Paek, J. Hoeflinger, and D. Padua, "Efficient and Precise Array Access Analysis," *ACM Transactions on Programming Languages and Systems*, vol. 24, no. 1, pp. 65–109, 2002.

[38] P. R. Panda, N. D. Dutt, and A. Nicolau, "Efficient Utilization of Scratch-Pad Memory in Embedded Processor Applications," in *EDTC '97: Proceedings of the 1997 European Conference on Design and Test*. IEEE Computer Society, 1997, p. 7.

[39] A. Ros, M. E. Acacio, and J. M. García, *Parallel and Distributed Computing*. IN-TECH, 2010, ch. Cache Coherence Protocols for Many-Core CMPs.

[40] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, N. Ranganathan, D. Burger, S. W. Keckler, R. G. McDonald, and C. R. Moore, "TRIPS: A Polymorphous Architecture for Exploiting ILP, TLP, and DLP," *ACM Transactions on Architecture and Code Optimization*, vol. 1, no. 1, pp. 62–93, 2004.

[41] S. Seo, J. Lee, and Z. Sura, "Design and Implementation of Software-Managed Caches for Multicores with Local Memory," in *HPCA '09: Proceedings of the 15th International Conference on High-Performance Computer Architecture*. IEEE Computer Society, 2009, pp. 55–66.

[42] S. Steinke, L. Wehmeyer, B. Lee, and P. Marwedel, "Assigning Program and Data Objects to Scratchpad for Energy Reduction," in *DATE '02: Proceedings of the conference on Design, Automation and Test in Europe*. IEEE Computer Society, 2002, pp. 409–415.

[43] M. Valero, M. Moreto, M. Casas, E. Ayguadé, and J. Labarta, "Runtime-Aware Architectures: A First Approach," *International Journal Supercomputing Frontiers and Innovations*, vol. 1, no. 1, pp. 29–44, 2014.

[44] J. Weinberg, M. O. McCracken, E. Strohmaier, and A. Snavely, "Quantifying Locality In The Memory Access Patterns of HPC Applications," in *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, 2005, pp. 50–62.

[45] R. P. Wilson and M. S. Lam, "Efficient Context-Sensitive Pointer Analysis for C Programs," in *PLDI '95: Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*. ACM, 1995, pp. 1–12.